Robust B$^+$-Tree-Based Indexing of Moving Objects

Christian S. Jensen Dalia Tiešytė Nerius Tradišauskas

Department of Computer Science
Aalborg University, Denmark
E-mail: {csj|dalia|nerius}@cs.aau.dk

Abstract

With the emergence of an infrastructure that enables the geo-positioning of on-line, mobile users, the management of so-called moving objects has emerged as an active area of research. Among the indexing techniques for efficiently answering predictive queries on moving-object positions, the recent B^x-tree is based on the B$^+$-tree and is relatively easy to integrate into an existing DBMS. However, the B^x-tree is sensitive to data skew. This paper proposes a new query processing algorithm for the B^x-tree that fully exploits the available data statistics to reduce the query enlargement that is needed to guarantee perfect recall, thus significantly improving robustness. The new technique is empirically evaluated and compared with four other approaches and with the TPR-tree, a competitor that is based on the R-tree. The results indicate that the new index is indeed more robust than its predecessor—it significantly reduces the number of I/O operations per query for the workloads considered. In many settings, the TPR-tree is outperformed as well.*

1 Introduction

The increasing interest in mobile-location data has served as motivation for the development of spatio-temporal indexes for the current and near-future positions of moving objects. Traditional spatial indexes designed for largely static data fall short in supporting mobile data. For static data, queries are much more frequent than updates. For mobile data, both updates and queries are frequent. In addition, such data represent continuous object movement. Velocity vectors are often utilized for predicting the near-future positions of such objects.

Two classes of spatio-temporal indexes have been proposed. A number of R*-tree-based [1] indexes store object positions in their native 2- or 3-dimensional space. Examples include the TPR-tree [10], the TPR*-tree [12], the STAR-tree [8], and the R$^{\mathrm{EXP}}$-tree [9]. Indexes in the other class employ data and query transformations to index object positions in "dual" spaces. Members of this class include STRIPES [7], which is based on the Quadtree [11], and the B^x-tree [2], which is based on the B$^+$-tree. STRIPES indexes n-dimensional positions in $2n$-dimensional space. The B^x-tree indexes n-dimensional positions in 1-dimensional space. Mokbel et al. [5] offer a brief, but quite comprehensive, survey of spatio-temporal indexes for the past, current, and near future positions of moving objects.

To ensure widespread use of a new index, easy integration into existing DBMSs is desirable. However, this includes complex integration with core DBMS components such as the query optimizer and the concurrency control and recovery sub-systems, which renders the integration very costly. Also, extensible technologies have so far proved inadequate for the integration of new indexes. It is thus attractive to be able to reuse an index that is already available in DBMSs when creating a new index that meets new needs. The B$^+$-tree is available in most DBMSs and has proved to be an efficient and versatile index with well-performing concurrency control and recovery mechanisms [4].

Following this line of reasoning, this paper proposes a B$^+$-tree-based index, termed the B$_r^x$-tree, for the indexing of the current and near-future positions of moving objects. This index extends the B^x-tree, which is relatively easy to integrate into a DBMS that already supports the B$^+$-tree [2]. The new index removes the reliance of the B^x-tree query transformation on global maximum and minimum object velocities, which leads to substantially improved query performance for settings with data skew, and thus results in a more robust index.

The paper defines and studies a range of possible query transformation algorithms, one of which minimally expands an argument query region while guaranteeing perfect recall. That is, all the objects that satisfy the query predicate are in the query result. The expansion is minimal in the sense that smaller expansions are not possible with the information available to the algorithm. The B$_r^x$-tree, which employs the optimal algorithm, represents a significant step toward a practical and versatile moving-object index. The B$_r^x$-tree is never worse than the B^x-tree, and experimental results show that it significantly outperforms the B^x-tree in some

settings.

The remainder of this paper is outlined as follows. Section 2 briefly describes the B^x-tree. Section 3 concerns query algorithms for the B^x-tree. Section 4 provides results of empirical studies of the algorithms covered and covers also the TPR-tree. Section 5 concludes and proposes directions for future research.

2 The B^x-Tree

The B^x-tree [2] adopts a transformation-based approach to the indexing of moving objects. Each object position, given as linear function from time to points in 2-dimensional space, is subjected to a transformation that uses a space-filling curve to map such functions to points in 1-dimensional space. These resulting points are then indexed using a B$^+$-tree. To ensure that queries have both perfect recall and are efficient, specific query transformations that counter the data transformation are used.

The B^x-tree may use any space-filling curve (as suggested by the "*x*" in the name). While the original experiments reported for the B^x-tree [2] use both Hilbert and Z-curves, this paper focuses on the Hilbert curve, as studies suggest that this curve has the better clustering properties [6].

2.1 Index Structure—Data Transformation

The data transformation in the B^x-tree is based on a partitioning of the time axis into equal-sized intervals. First, a problem parameter Δt_{mu}, the *maximum update interval*, is introduced. Most objects are expected to be updated within the duration of Δt_{mu}. The time axis is first partitioned into intervals of this length. Each of the resulting intervals is then further partitioned into n equal-length intervals, called phases. An index partition, or sub-tree of the B$^+$-tree, is subsequently created for each phase. At most $n+1$ partitions exist at any time. The reference time t_r^i of phase i is the start time of the phase, $i = 1, \ldots, n+1$.

When an object issues an update at time t_{up}, the old index key recorded for the object is removed, and a new one is inserted. For the insertion, the partition that intersects with t_{up} is identified, and the position of the object as of the reference time of that partition is used for the subsequent indexing in the B$^+$-tree. The insertion is made into the identified partition by prefixing the index value that will be computed with the number of the partition.

As time passes and updates arrive, old index keys are removed from old partitions and new ones are inserted into new partitions. The lifetime of a partition is $\Delta t_{mu} + \Delta t_{mu}/n$, after which time it expires. Any index key that might remain in a partition at the time of expiry is "reinserted" using the reference time t_r^i of the most recently created partition i.

The data space is partitioned into a uniform grid that specifies the granularity of the data space. Positions are mapped to grid cells, and the cells are enumerated using a space-filling curve. This way, a 2-dimensional point is mapped to a 1-dimensional point. An important requirement to a space-filling curve is that it offers a high level of clustering: cells close in 2-dimensional space should be close in the 1-dimensional space to which they are mapped.

The space partitioning in the B^x-tree is illustrated in Figure 1, where the Hilbert curve is used.

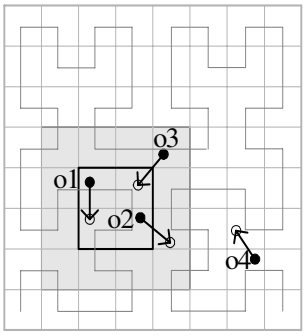

Figure 1. Space partitioning and query expansion in the B^x-tree

2.2 Query Transformation

A timeslice query, which retrieves all objects that are within a given rectangle at a given point in time, must check each existing partition for qualifying objects. In one partition, the query region is first expanded by a factor of the current maximum, $\vec{V}_{max}$, and minimum, $\vec{V}_{min}$, projections of the velocities $\vec{v}$ of all objects: $\vec{V}_{max} = (\max_{\vec{v} \in V}\{v^x\}, \max_{\vec{v} \in V}\{v^y\})$, $\vec{V}_{min} = (\min_{\vec{v} \in V}\{v^x\}, \min_{\vec{v} \in V}\{v^y\})$, where v^x and v^y are the projections of velocity $\vec{v}$ onto the x and y axes, and V is the set of velocities of the currently indexed objects. Next, the expanded query rectangle may be reduced if the maximum and minimum velocities of the objects that fall into the expanded query region are smaller than $\vec{V}_{max}$ or larger than $\vec{V}_{min}$. These velocities are stored for each cell and each partition in a main-memory *velocity histogram*.

A filter-and-refinement procedure is used to guarantee a correct query answer. The index returns all objects that may qualify, upon which the linear functions of these are used for refinement.

In Figure 1, the *shaded region* is the expanded query region for the *smaller rectangle* inside the shaded region, which is the original query rectangle. The *filled circles* denote object positions at the reference time of the phase, and the *empty circles* are the positions at the query time. The *arrows* denote the velocity vectors of the objects. Object $o1$ is inside the original query region and stays there until the query time. Object $o2$ is first inside the query region, but

has moved out by the query time. Object $o3$ is first outside the query region, but is inside by the query time. The query result is $\{o1, o3\}$.

3 New Query Processing Algorithms

Being based on the B$^+$-tree, the B^x-tree has potential for being integrated into DBMSs that already support the the B$^+$-tree. However, the current query expansion algorithm does not fully use the available information. We thus proceed to propose a solution that fully exploits the information available in the index, the objective being to obtain a better-performing and more robust index.

Three query types (defined in [10]) are analyzed. Let t_q, t_q^1, and t_q^2 be time points and let q_r, q_r^1, and q_r^2 be 2-dimensional rectangles.
Timeslice query $Q = (q_r, t_q)$ returns the objects that intersect with q_r at time t_q.
Window query $Q = (q_r, t_q^1, t_q^2)$ returns the objects that intersect with q_r at some time during time interval $[t_q^1, t_q^2]$.
Moving window query $Q = (q_r^1, q_r^2, t_q^1, t_q^2)$ returns the objects that intersect, at some time during $[t_q^1, t_q^2]$, with the trapezoid obtained by connecting rectangles q_r^1 and q_r^2 at times t_q^1 and t_q^2, respectively.

3.1 Query Expansion Approaches in the B^x-Tree

The B^x-tree uses an algorithm that reduces the initial, expanded query region somewhat conservatively—the maximum and minimum velocities in the velocity histogram can be used more aggressively.

The information available in main memory for the query expansion algorithm consists of the global maximum ($\vec{V}_{max}$) and minimum ($\vec{V}_{min}$) velocities, and the velocity histogram. The latter stores, for each cell of the data space, the maximum and minimum velocities of the objects that fall inside the cell. When an update occurs, the velocity histogram is also updated conservatively. In particular, the cell within which a new position falls is updated, but the cell from which the old position is removed is not updated. Therefore, some cells may store expired values.

The maintenance cost of the velocity histogram depends on the granularity of the data space. When a new index partition is created, the cells of the velocity histogram for this partition are initialized. This requires $O(n)$ operations, where n is the number of cells. An update of one (memory resident) histogram cell requires only constant processing time. If there is a need to reduce the histogram maintenance cost, it is possible to modify its granularity so that one histogram cell corresponds to a group of data space cells. However, this may adversely affect query I/O performance.

To offer a better understanding of the possibilities for query expansion, we define five approaches.

Naive Approach The original query region is simply expanded according to the global maximum and minimum velocities. The velocity histogram is not used.

Semi-Naive Approach This approach extends the naive approach by using the velocity histogram once to find the local maximum and minimum velocities within the naively expanded query region, possibly reducing the query region. This method is used in [2].

Iterative Approach First, the semi-naive approach is applied. Then the query region is reduced iteratively according to the new maximum and minimum velocities from the reduced query region until no reduction occurs.

Optimal Approach The information in the velocity histogram is utilized fully. Each histogram cell intersected by the query following a naive expansion is examined separately. If this cell may contain qualifying objects, the index cell is retrieved from disk.

Ideal Approach Only those cells that actually contain objects that qualify for the query are selected. It is not possible to implement this approach with the information available. This approach reports the minimum number of cells that must be covered by an expanded query and is included for comparison purposes only. It enables us to determine how close the practical approaches get to the ideal expansion.

The iterative approach for timeslice queries is detailed in Algorithm 1. As the iterative approach extends the naive and semi-naive approaches, these approaches are explained as parts of the iterative approach. The naive approach executes only line 1 of Algorithm 1. The query region is expanded by a factor of global maximum and minimum velocities. The semi-naive approach applies lines 3–4 only once. The maximum and minimum velocities among all velocities in the initially expanded region are obtained from the velocity histogram VH (line 3). Then, the query region is expanded using these velocities (line 4). The iterative approach repeats lines 3 and 4 until it is not possible to further reduce the expanded region.

Algorithm 1: Iterative query expansion

Input: Query rect. q_r, query time t_q, ref. time t_r
1: $eq_r \leftarrow expand(q_r, t_q, t_r, \vec{V}_{max}, \vec{V}_{min})$
2: **repeat**
3: $(\vec{v}_{max}, \vec{v}_{min}) \leftarrow getVelocities(eq_r, VH)$
4: $eq_r \leftarrow expand(q_r, t_q, t_r, \vec{v}_{max}, \vec{v}_{min})$
5: **until** $(\vec{v}_{max}, \vec{v}_{min}) = getVelocities(eq_r, VH)$
6: **return** eq_r

As the dependency on the global parameters $\vec{V}_{max}$ and $\vec{V}_{min}$ decrease, the query expansion becomes increasingly robust to data skew. The numbers of cells selected by the naive approach depend on the current maximum and minimum velocities of the objects in the entire data space. The numbers of cells selected by the semi-naive and iterative approaches depend on the current maximum and minimum velocities of the objects in and around the query region.

The numbers of cells selected by the optimal and ideal approaches are independent of the global maximum and minimum velocities. We proceed to explain the optimal approach in more detail.

3.2 Optimal Query Expansion

The optimal query expansion for timeslice queries is defined in Algorithm 2. The algorithm takes a query rectangle q_r and a query time t_q as input. The set of cells CS that

Algorithm 2: Optimal query expansion

Input: Query rect. q_r, query time t_q, ref. time t_r
Output: Set of cells CS
1: $CS \leftarrow \emptyset$
2: $eq_r \leftarrow expand(q_r, t_q, t_r, \vec{V}_{max}, \vec{V}_{min})$
3: **for all** cells c such that $c \cap eq_r \neq \emptyset$ **do**
4: **if** $canReach(c, q_r, t_q, t_r)$ **then**
5: $CS \leftarrow CS \cup \{c\}$
6: **end if**
7: **end for**
8: **return** CS

the algorithm returns is initialized to an empty set (line 1). The query is first expanded using the current maximum and minimum velocities and the query time (line 2). Then each cell c that overlaps with the expanded query region is examined (lines 3–7). Function $canReach$ (line 4, defined in Algorithm 4) checks whether an object in a cell c may qualify for the query by checking for an overlap between the expanded cell and the initial query region. If the condition is true, the cell is included in the result, CS (line 5). The cells in CS must be read from disk.

Function $expand$, defined in Algorithm 3, is used in the

Algorithm 3: function $expand$

Input: Query rect. $[qp_l, qp_u]$, query time t_q, ref. time t_r, velocities $\vec{v}_{max}$, $\vec{v}_{min}$
Output: Expanded query rectangle $[eqp_l, eqp_u]$
1: **if** $t_q \geq t_r$ **then**
2: $eqp_l \leftarrow qp_l - \vec{v}_{max}(t_q - t_r)$
3: $eqp_u \leftarrow qp_u - \vec{v}_{min}(t_q - t_r)$
4: **else**
5: $eqp_l \leftarrow qp_l - \vec{v}_{min}(t_q - t_r)$
6: $eqp_u \leftarrow qp_u - \vec{v}_{max}(t_q - t_r)$
7: **end if**
8: **return** $[eqp_l, eqp_u]$

naive, semi-naive, iterative, and optimal approaches. The query region is defined by its lower left corner, qp_l, and upper right corner, qp_u. It is expanded in all directions by a factor of the maximum and minimum velocities multiplied by the difference between the query time and the partition reference time. If the partition reference time is earlier than the query time, the query region is expanded using the opposite directions of velocities, termed a *backward expansion* (lines 1–3). Otherwise, a *forward expansion* using the actual velocities is applied (lines 4–6).

Function $canReach$, in Algorithm 4, is only used in the optimal approach. The naive algorithm stops after using function $expand$. The semi-naive and iterative approaches reduce the query further, but do not consider each cell separately. In function $canReach$, the maximum and minimum

Algorithm 4: function $canReach$ for timeslice queries

Input: Cell $[cp_l, cp_u]$, query rect. q_r, query time t_q, ref. time t_r
Output: $canReach \in \{\textbf{true}, \textbf{false}\}$
1: $(\vec{v}_{max}, \vec{v}_{min}) \leftarrow getVelocities(c, VH)$
2: $[ecp_l, ecp_u] \leftarrow expandCell([cp_l, cp_u], t_q, t_r, \vec{v}_{max}, \vec{v}_{min})$
3: **return** $[ecp_l, ecp_u] \cap q_r \neq \emptyset$

velocities $\vec{v}_{max}$, $\vec{v}_{min}$ for the cell are obtained from the velocity histogram VH (line 1). A cell is defined by its lower left corner cp_l and upper right corner cp_u. It is expanded according to its maximum and minimum velocities by calling function $expandCell$ (line 2). If the expanded cell overlaps with the query region (line 3), objects in the cell may be able to reach the query region at the time of the query.

Function $expandCell$ is given in Algorithm 5. Forward (lines 1–3) or backward (lines 4–6) expansion are used, depending on the query time. The maximum and minimum

Algorithm 5: function $expandCell$

Input: Cell $[cp_l, cp_u]$, query time t_q, ref. time t_r, velocities $\vec{v}_{max}$, $\vec{v}_{min}$
Output: Expanded cell $[epc_l, epc_u]$
1: **if** $t_q \geq t_r$ **then**
2: $ecp_l \leftarrow cp_l + \vec{v}_{min}(t_q - t_r)$
3: $ecp_u \leftarrow cp_u + \vec{v}_{max}(t_q - t_r)$
4: **else**
5: $ecp_l \leftarrow cp_l + \vec{v}_{max}(t_q - t_r)$
6: $ecp_u \leftarrow cp_u + \vec{v}_{min}(t_q - t_r)$
7: **end if**
8: **return** $[epc_l, epc_u]$

velocities of a cell are multiplied by the difference between the query time and the partition reference time.

The algorithms presented so far are for timeslice queries. The algorithms for window and moving window queries are similar. For these queries, functions $expand$ and $canReach$ use a time interval $[t_q^1, t_q^2]$ instead of a time point t_q; and for the moving window queries, two rectangles (q_r^1, q_r^2) are used instead of q_r.

Function $canReach$ for window queries is given in Algorithm 6. Query expansion is performed with the start and end query times (lines 2 and 3). A union of both expanded regions is taken (line 4). Function $expand$ uses the same principle.

For moving window queries, function $expand$ must select all cells that overlap with the moving query. The rectan-

gle of the expanded query region is the minimum bounding rectangle that includes query rectangles q_r^1 and q_r^2.

Algorithm 6: function *canReach* for window queries

Input: Cell $[cp_l, cp_u]$, query rect. q_r, query time $[t_q^1, t_q^2]$, ref. time t_r
Output: $canReach \in \{\textbf{true}, \textbf{false}\}$
1: $(\vec{v}_{max}, \vec{v}_{min}) \leftarrow getVelocities(c, VH)$
2: $[ecp_l^1, ecp_u^1] \leftarrow expandCell([cp_l, cp_u], t_q^1, t_r, \vec{v}_{max}, \vec{v}_{min})$
3: $[ecp_l^2, ecp_u^2] \leftarrow expandCell([cp_l, cp_u], t_q^2, t_r, \vec{v}_{max}, \vec{v}_{min})$
4: $[ecp_l, ecp_u] \leftarrow [ecp_l^1, ecp_u^1] \cup [ecp_l^2, ecp_u^2]$
5: **return** $[ecp_l, ecp_u] \cap q_r \neq \emptyset$

Function *canReach* for moving window queries is more complex. Query velocity vectors are defined for the lower and upper corners of the query rectangle. The velocity values are chosen so that the corner points of query rectangle q_r^1 move to the corresponding points of rectangle q_r^2 in time $t_q^2 - t_q^1$. If the query is moving, but the rectangle is not changing its shape, the query velocity in time period $[t_q^1, t_q^2]$ is subtracted from the cell's velocity. Otherwise, the lower and upper corners of the query rectangle are considered separately in the inclusion test. The detailed algorithm is omitted due to space limitations.

Figure 2 shows an example of the practical query expansion approaches. The *dark-shaded* 2×2 rectangle is the

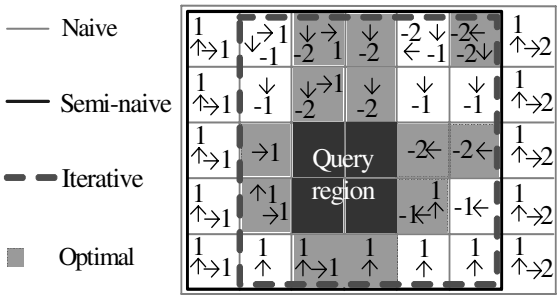

Figure 2. Four query expansion approaches

original query region. Assume that, with velocity 1 in one dimension, a distance of one cell width/height is traveled until the query time. The maximum and minimum velocities of each cell are shown. The velocities that are not given in the figure are equal to 0. The current maximum and minimum velocities are: $\vec{V}_{max} = (2, 1)$, $\vec{V}_{min} = (-3, -2)$. The naive approach expands the query to a 7×5 *cell rectangle*. In the other approaches, this is the initial expansion of the query, obtained by function *expand*.

The semi-naive approach calculates the maximum and minimum velocities of the expanded query region, $\vec{v}_{max} = (2, 1)$, $v_{min} = (-2, -2)$. The query region is then reduced by the 5 rightmost cells. The iterative approach continues to reduce the query region by calculating new maximum and minimum velocities: $\vec{v}_{max} = (1, 1)$ and $v_{min} = (-2, -2)$. As a result, the query region is further reduced by the 5 leftmost cells. After this, the query expansion cannot be reduced, as the maximum and minimum velocities do not change. The optimal approach considers each cell separately. Only the *lightly shaded* cells and the cells that are covered by the original query region must be read from disk and checked for qualifying objects.

Lemma. *With the available information, the optimal approach yields the minimum possible query enlargement.*

Proof: A conceptual proof for timeslice queries follows.

The available information about an object's position is a cell with coordinates $((cp_l^x, cp_l^y), (cp_u^x, cp_u^y))$. The actual position in the cell is unknown. At reference time t_r, the object's position is given as follows:

$$(p^x, p^y)\ ,\ cp_l^x \leq p^x \leq cp_u^x \wedge cp_l^y \leq p^y \leq cp_u^y \quad (1)$$

We must determine whether the object may appear in the query region at the query time.

The available information about the object's velocity is the maximum and minimum velocities of the cell, as stored in the velocity histogram: $\vec{v}_{max} = (v_{max}^x, v_{max}^y)$ and $\vec{v}_{min} = (v_{min}^x, v_{min}^y)$. Let time duration Δt_{diff} be the difference between the query time q_t and the partition reference time t_r as of which the object positions are stored. Assume that $t_q > t_r$. The opposite case is symmetric. The actual object velocity during time Δt_{diff} is given by:

$$(v^x, v^y)\ ,\ v_{min}^x \leq v^x \leq v_{max}^x \wedge v_{min}^y \leq v^y \leq v_{max}^y \quad (2)$$

At query time t_q, the object's position is given by:

$$(p_q^x, p_q^y) = (p^x + \Delta t_{diff}\, v^x, p^y + \Delta t_{diff}\, v^y) \quad (3)$$

By substituting velocities v^x, v^y by the inequalities in expression (2) and positions p^x, p^y by the inequalities in expression (1), the following inequalities are obtained:

$$cp_l^x + \Delta t_{diff}\, v_{min}^x \leq p^x + \Delta t_{diff}\, v^x \leq cp_u^x + \Delta t_{diff}\, v_{max}^x$$
$$cp_l^y + \Delta t_{diff}\, v_{min}^y \leq p^y + \Delta t_{diff}\, v^y \leq cp_u^y + \Delta t_{diff}\, v_{max}^y$$

At time t_q, the point (p_q^x, p_q^y) belongs to the rectangle:

$$(ecp_l, ecp_u) = ((cp_l^x + \Delta t_{diff}\, v_{min}^x, cp_l^y + \Delta t_{diff}\, v_{min}^y),$$
$$(cp_u^x + \Delta t_{diff}\, v_{max}^x, cp_u^y + \Delta t_{diff}\, v_{max}^y)).$$

Rectangle (ecp_l, ecp_u) is identical to the expanded cell in Algorithm 4. The cell position and velocities utilize all available information. The conservative maximum and minimum velocities allow the algorithm to achieve perfect recall. The actual velocity of an object is always bounded by the maximum and minimum velocities of the corresponding cell. If the expanded cell overlaps with the query rectangle, this means that an object in the cell may be in the query region at the query time. As a result, it is not possible to eliminate cells selected by the optimal algorithm. □

The proofs for window and moving window queries are similar. Query time t_q becomes a time interval $[t_q^1, t_q^2]$, and $\Delta t_{diff} \in [t_q^1 - t_r, t_q^2 - t_r]$. Thus, the object position belongs to the region of the union of cell expansions obtained with t_q^1 and t_q^2. For moving window queries, the cell expansion is different for each time point between the query start and end times, as the query region is different as well. A moving window query may be considered as a union of timeslice queries in time interval $[t_q^1, t_q^2]$ that have different query regions for each time point.

The cells selected by the optimal algorithm are always a subset of the cells selected by the naive, semi-naive, or iterative approaches. As a result, the optimal algorithm will never require more I/O than the other algorithms.

The improved query expansion is promising because queries require significantly more I/O than do updates [2, 3] and because the current B^x-tree query expansion algorithm is not optimal. We note that the algorithm for the optimal approach may also be used in other spatio-temporal indexes that employ query expansion.

4 Empirical Study of the Query Expansion Algorithms

This section reports on an empirical study with variants of the B^x-tree that use the different query expansion approaches defined in Section 3.1. The index that employs the optimal approach is denoted as the B_r^x-tree.

4.1 Experimental Settings

The indexes with the different query-expansion algorithms are evaluated and compared using the COST benchmark [3] as well as additional experiments. Experimental results for the TPR-tree [10] performance are provided for comparison. The TPR-tree is a robust index that is also a predecessor of a number of other spatio-temporal indexes.

The COST benchmark simulates moving-object scenarios and is designed specially with the evaluation of moving-object indexes in mind. The benchmark includes a workload generator that, based on settings for a number of parameters, generates workloads that intermix updates and queries.

The generator, which extends the one used by Šaltenis et al. [10], assumes that object movement is either random or network-based. To accommodate the latter, a number of "hubs" with random positions are generated connected with line segments to produce a complete, bi-directional, spatial graph. A constant number of objects then move between hubs until the end of a simulation.

The maximum speed of an object is chosen at random from a set of maximum speeds, and an object accelerates, travels at constant speed, and then decelerates when moving from one hub to another.

The workloads used in the experiments reported on here are generated using the following default parameter settings. A total of 500 randomly positioned hubs are used. The total data space is 100×100 km. There are 100 K objects that receive a total of 200 K updates. Queries have an average spatial extent of 5×5 km. Two queries are executed for each 400 updates, meaning that a total of 1 K queries are executed. The disk page size is 1 K. A main-memory LRU buffer is used that has a capacity of 50 pages.

We have found it useful to customize the COST benchmark specification slightly. Specifically, each object is always updated within the maximum intervals of 120 s, in contrast to the original unpredictable maximum update interval used in the benchmark. The experiments thus use a time-based update policy where the average time duration in-between updates to an object is 60 s. Experiments are performed with partitionings of the data space into 512×512, 1024×1024, and 2048×2048 equal-sized cells of sizes 1×1, 1/2×1/2, and 1/4×1/4 km.

The indexes used have 2 phases, meaning that at most 3 partitions coexist at a single point in time.

The experiments consider the query performance of the five index variants. The average numbers of I/O operations per query and the numbers of cells retrieved are reported. When the queries are distributed uniformly in space and the number of queries approaches infinity, the average number of objects retrieved is proportional to the number of cells retrieved.

Studies on indexes typically show that the CPU performance, which we do not consider directly, is strongly correlated with the I/O performance. The update performance is not considered, as this does not differ among the variants.

4.2 Experimental Results

We proceed to consider selected results from the COST benchmark and then consider results of additional experiments designed specifically for this problem.

4.2.1 COST Benchmark Experiments

Figure 3 shows the results of COST benchmark Experiment 3.1. Objects are assigned either speed 25 m/s or 200 m/s. A total of 11 workloads are generated so that the fractions of objects with speed 200 m/s are: $0.02, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.98$. The figure shows the average number of cells retrieved (in units of 1,000 cells) and the average I/O per query for the three different space granularities.

The naive approach selects a significantly higher number of cells compared to the other approaches, as the expanded query may overlap with the cells that are outside the space in which the generated positions of objects appear. Only the naive approach cannot eliminate these cells. The semi-naive

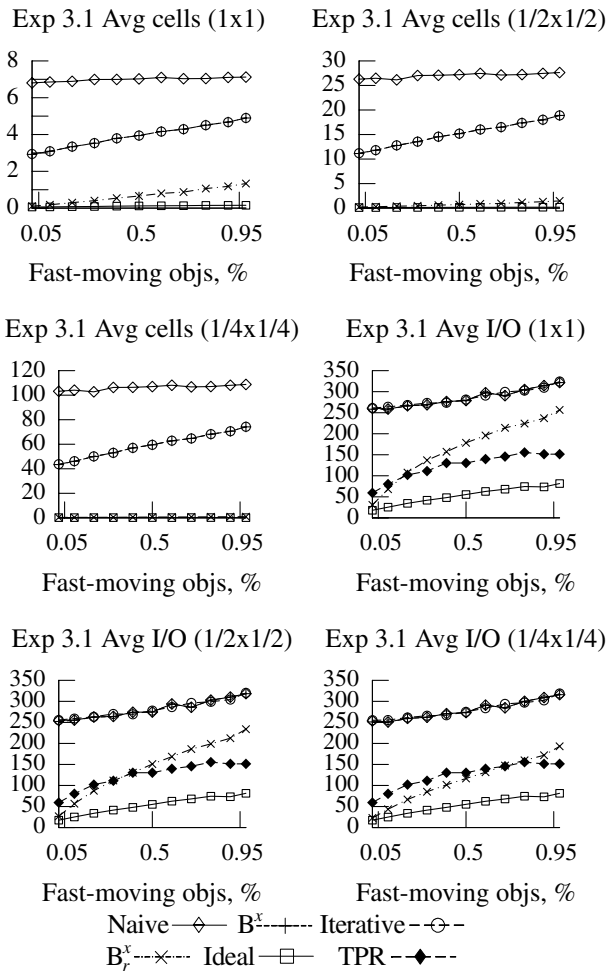

Figure 3. Varying fast-moving objects

and iterative approaches perform similarly. The optimal approach is significantly better than the other implementable approaches, and it is close to the ideal approach. Other experiments show that the numbers of cells selected exhibit little sensitivity to position and velocity skew.

The I/O performance of the naive, semi-naive, and iterative approaches are similar to each other (the curves overlap). The average selectivity is 250 objects per query. With 2% of fast-moving objects, query result contains an average of 5 fast-moving objects. The expanded queries include even more fast-moving objects and are unlikely to be reduced by the semi-naive and iterative approaches.

For low amounts of fast-moving objects, the I/O performance of the B^x_r-tree is close to the ideal, but increases as the percentage of fast-moving objects increases. This is because the expansions of cells that contain fast-moving objects are large.

The B^x_r-tree is the most sensitive to the cell size, compared to the other approaches. As the cell size decreases, yielding a more accurate velocity histogram, the performance improves.

The TPR-tree is outperformed by the B^x_r-tree when the amount of fast-moving objects is low, and especially when the cell size is small. With a $1/4 \times 1/4$ km cell size, the performance of the B^x_r-tree is close to the performance of the TPR-tree, even for high amounts of fast-moving objects. The naive, semi-naive, and iterative approaches, however, require significantly more I/O than the TPR-tree.

Figure 4 reports on additional COST benchmark results. The figure presents results of Experiments 1, varying the number of objects from 100 to 1000 K in increments of 100 K; 2.2, varying the number of hubs from 20 to 200 in increments of 20 hubs; 3.3, varying the maximum speeds from 30 to 300 m/s in increments of 30 m/s; 5.3, varying the average update interval from 2 to 20 min in increments of 2 min; 7.2, varying the times of queries relative to the current time from 0 to 120 s in increments of 20 s; and 8.1, varying the spatial extents of queries from 0.05 to 0.95% of the space in increments of 0.1%. I/O costs are reported for the B^x-tree variants with cell sizes 1×1 and $1/4 \times 1/4$ km, and for the TPR-tree.

As can be seen, the B^x_r-tree significantly outperforms the three other implementable approaches. The improvements over these are especially high when the space granularity is coarse. The differences among the naive, semi-naive, and iterative approaches are slight because all objects have similar speeds. Only when there is a high number of objects, the performance of the naive approach degrades more than for others due to the larger index size and larger numbers of objects selected.

The B^x_r-tree outperforms the TPR-tree in default settings, when the future times of queries are large (over 60 s), when the query regions are large (over 0.5% of the space), and especially when there are large amounts of objects. In the other cases, the query I/O performance of the B^x_r-tree with cell sizes $1/4 \times 1/4$ km is comparable to that of the TPR-tree.

4.2.2 Additional Experiments

Most of the benchmark experiments do not yield significant differences in I/O performance among the naive, semi-naive, and iterative approaches. We thus proceed to report on additional experiments that elicit the settings when these approaches exhibit performance differences.

The first experiment varies the percentage of fast-moving objects from 0.1 to 1% in increments of 0.1%. The other settings are as in Experiment 3.1. The experimental results of the TPR-tree are not reported, as these experiments aimed to compare the different variants of the B^x-tree. Figure 5 reports the I/O performance for this and the next experiment.

The second experiment involves 8 workloads. The set

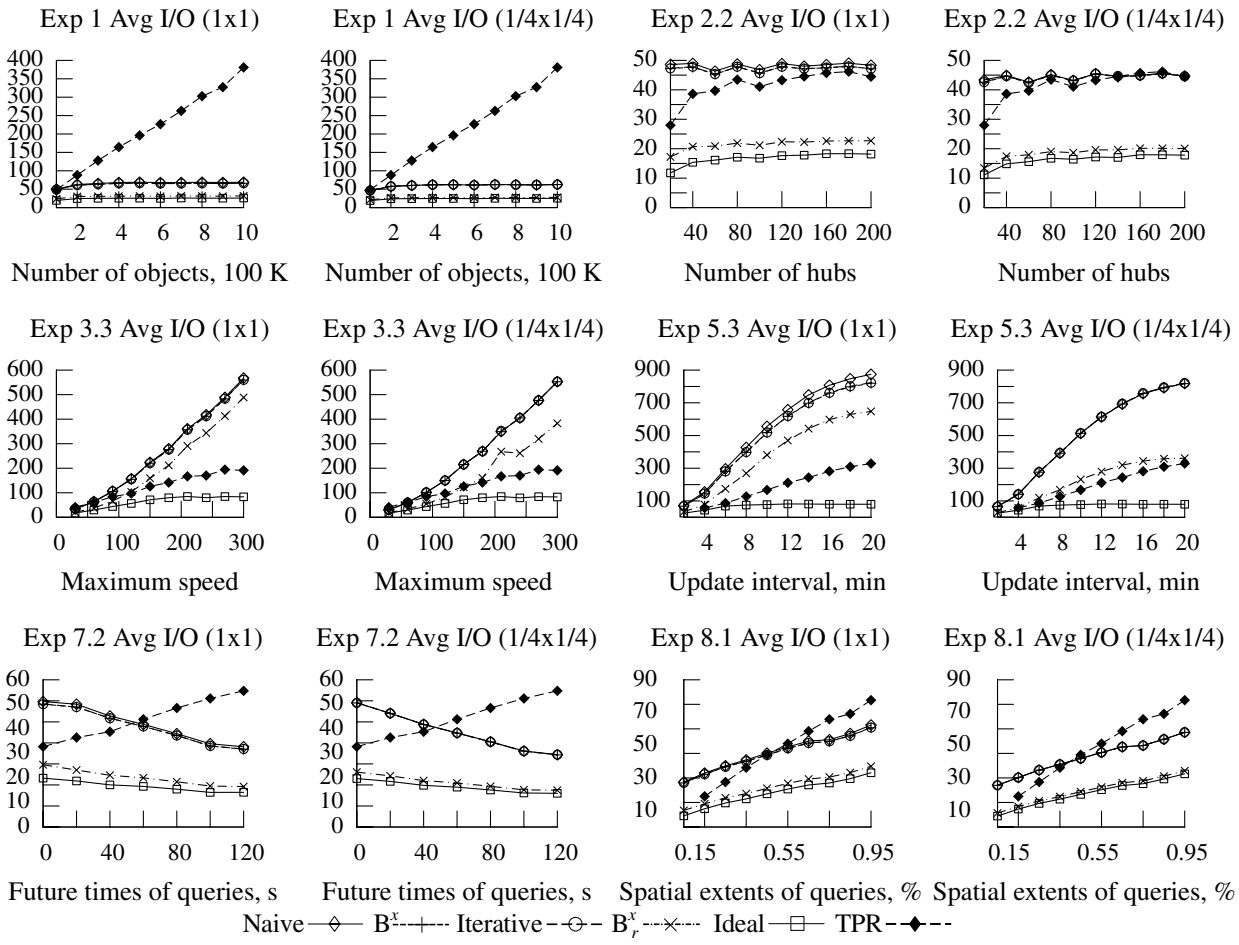

Figure 4. COST benchmark Experiments 1, 2.2, 3.3, 5.2, 7.2, and 8.1

$\{12.5, 25, 50, 100, 150, 200\}$ of maximum speeds (in m/s) is used. The probability of speed 12.5 m/s varies from 99.96 to 99.68% in decrements of 0.04%. The probabilities of the other speeds vary from 0.01 to 0.08% in increments of 0.01%.

For 0.1–0.3% fast-moving objects in the first experiment, the I/O performance of the naive approach is visibly worse than for the semi-naive and iterative approaches. The differences between the semi-naive and iterative approaches remain slight.

Due to the very few fast-moving objects in the second experiment, the differences between the naive and the other approaches are substantial. The naive approach depends on global maximum and minimum velocities, which deviate substantially from the actual velocities.

The iterative approach in the second experiment slightly outperforms the semi-naive approach. When the fraction of fast-moving objects increases, these approaches exhibit degrading performance. Only the B^x_r-tree is unaffected, and its performance is almost equal to the ideal case.

The I/O performance is largely insensitive to different space granularities and only improves slightly as the cell size decreases.

5 Conclusions and Research Directions

Being relatively easy to integrate into existing DBMSs is a desirable property of a new index. However, integrating an index into a DBMS is a resource intensive and challenging endeavor. This renders it particularly relevant to attempt to reuse indexes that are already supported by DBMSs, e.g., the B^+-tree, when designing a new spatio-temporal index.

Using advanced data and query transformations, the B^x-tree exploits the B^+-tree to enable the indexing of the current and future positions of moving objects. This paper observes that the query performance of the B^x-tree is vulnerable to data skew and proposes an improved query algorithm.

The paper defines and studies four query transformations that may be applied with the B^x-tree. These differ in their use of an available velocity histogram. In addition,

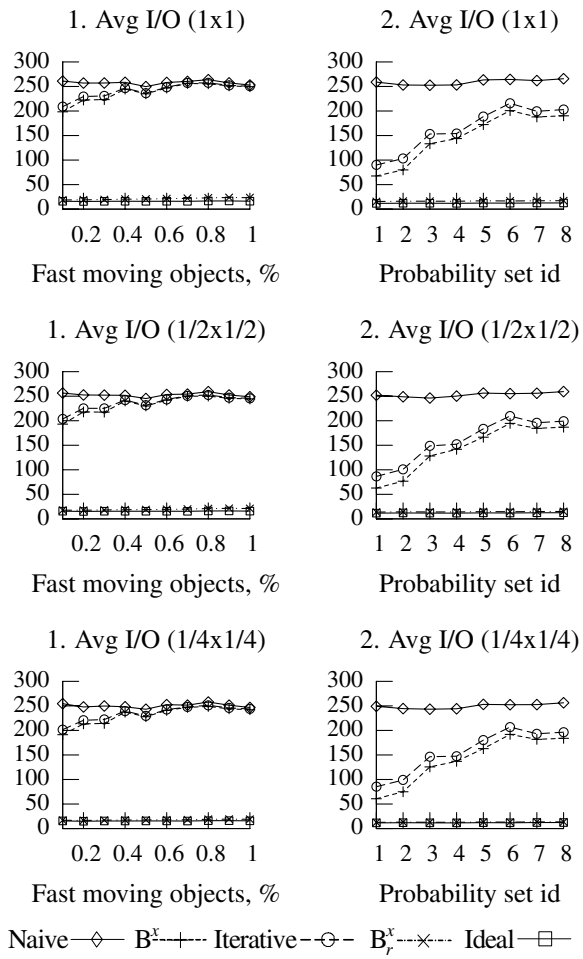

Figure 5. Few fast-moving objects

an "ideal" transformation that uses more information than what is available is included, the objective being to determine how close to the ideal the former transformations are.

One of the four transformations uses the velocity histogram optimally and offers a significant reduction of the query region compared to the original B^x-tree transformation. In contrast to the other transformations, it is independent of the global maximum and minimum velocities of the objects. Further, because the cells selected for querying by the B_r^x-tree, which employs the optimal transformation, is a subset of the cells selected by the B^x-tree, the B^x-tree never outperforms the B_r^x-tree.

For the workloads considered, the B_r^x-tree exhibits substantial query performance improvements over the B^x-tree, and the experimental study also shows that the I/O performance of the B_r^x-tree in many cases is close to the "ideal." The improvements are especially high for workloads with few fast-moving objects. In many cases, the B_r^x-tree also outperforms the TPR-tree.

This work may be continued in several directions. In particular, it is highly relevant to try to reduce the reliances of transformation-based indexes such as the B^x-tree on static index parameters. First, it would be attractive to use a dynamic maximum update interval instead of the currently used static value. The performance of the B^x-tree may degrade when the maximum update interval is smaller than the actual interval in-between consecutive updates of an object. Second, it would be interesting to consider the use of a number of phases that changes over time and adapts to workloads that change across time (e.g., day versus night behavior). The number of phases should adapt to the time windows of queries and arrival rates of updates. Third, it might be attractive to use a data space granularity that is dynamic and space-varying. The granularity should adapt to changing workloads. Fourth, it may be attractive to store additional information in the velocity histogram, as this may further reduce the query expansion.

Acknowledgments This research was conducted within the project Telematics Applications Based on Ubiquitous Sensor Networks, funded by ETRI, South Korea. In addition to his primary affiliation with Aalborg University, C. S. Jensen is an adjunct professor in Department of Technology, Agder University College, Norway.

References

[1] A. Guttman. R-Trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pp. 47–57, 1984.

[2] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *Proc. VLDB*, pp. 768–779, 2004.

[3] C. S. Jensen, D. Tiešytė, and N. Tradišauskas. The COST benchmark—comparison and evaluation of spatio-temporal indexes. In *Proc. DASFAA*, 2006.

[4] T. Johnson and D. Shasha. The performance of current data structure algorithms. In *ACM TODS*, pp. 51–101, 1993.

[5] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE DEBull.*, 26(2): 40–49, 2003.

[6] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE TKDE*, 13(1): 124–141, 2001

[7] J. M. Patel, A. Arbor, Y. Chen, and V. P. Chakka. STRIPES: an efficient index for predicted trajectories. In *Proc. ACM SIGMOD*, pp. 635–646, 2004.

[8] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-tree: an efficient self-adjusting index for moving objects. In *Proc. ALENEX* (LNCS 2409), pp. 178–193, 2002.

[9] S. Šaltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proc. IEEE ICDE*, pp. 463–472, 2002.

[10] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD*, pp. 331–342, 2000.

[11] H. Samet. The Quadtree and related hierarchical data structures. *ACM Comp. Surv.*, 16(2): 187–260, 1984.

[12] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, pp. 790–801, 2003.